Title: SG5 Transactional Memory Support for C++ Update

Number: N4180
Date: 2014-10-10
Authors: Michael Wong, michaelw@ca.ibm.com
Victor Luchangco, victor.luchangco@oracle.com
with other members of the transactional memory study group (SG5), including:
Hans Boehm, hboehm@google.com
Justin Gottschlich, justin.gottschlich@intel.com
Jens Maurer, jens.maurer@gmx.net
Paul McKenney, paulmck@linux.vnet.ibm.com
Maged Michael, magedm@us.ibm.com
Mark Moir, mark.moir@oracle.com
Torvald Riegel, triegel@redhat.com
Michael Scott, scott@cs.rochester.edu
Tatiana Shpeisman, tatiana.shpeisman@intel.com
Michael Spear, spear@cse.lehigh.edu
Project: Programming Language C++, SG5 Transactional Memory
Reply to: Michael Wong michaelw@ca.ibm.com (Chair of SG5)
For: EWG, LEWG
Revision: 2

Introduction

Transactional memory supports a programming style that is intended to facilitate parallel execution with a comparatively gentle learning curve. This document further describes a proposal developed by SG5 to introduce transactional constructs into C++ as a Technical Specification based on N3999.

This paper (N4180) is an update of the Transactional Memory design proposal outlined in N3999 based on feedbacks from Rapperswil Core and Library working group Review. A separate paper (N4179) describes the wording changes that support all the following changes.

Specifically, it describes these changes to N3999's design.

1. Request in Rapperswil CWG review for static checking for the replace code-bloat issue from safe-by-default. This is already approved by EWG in Rapperswil.
2. Add new function attribute `[[optimized_for_synchronized]]` for optimized invocation inside synchronized statement to enable speculation, as a consequence of the static checking. This is pending approval by EWG.
3. Add keyword `transaction_safe noinherit` for virtual functions to make transaction-safety not be viral in the virtual functions of the derived class to address LWG review in Rapperswil on transaction-safety of destructors of exception types derived from std::exception. This is pending approval by EWG.
4. Adds Transaction-safety for all containers and iterator-related functions. This is pending approval by LEWG.
5. Adds `tx_exception`. This is pending approval by LEWG.

Changes in previous revisions

- N3999: From N3919, we updated with discussion results from EWG, and the TM Evening Session of Issaquah. We considered the memory model effect from Chandler on whether a transaction that does not access any shared data induces any happens-before arcs – and in particular whether transactions that are provably thread-local must incur synchronization costs. We also added Core Standard Wording to support TM TS in preparation for the Draft Document. Core Standard Wording will be reviewed in a separate telecon call on June 2, and likely again at Rapperswil. Library wording is posted here and in N4000 and was reviewed in Rapperswil.
- N3919: From N3859, this was the EWG approved version that was also voted in Issaquah as the indicated document for the TM TS NP.
  We added examples, and modified the syntax as directed by EWG vote.
- N3859: From N3718, we made several changes based on feedback and discussion at and since the Chicago meeting, including:
  – The atomic transactions and relaxed transactions of N3718 were renamed atomic blocks and synchronized blocks respectively (and the relevant keywords were changed to reflect this).
  – Synchronized blocks are defined before and without reference to atomic blocks.
  – We eliminated the use of escape to refer to an exception being thrown but not caught within a transaction (and the relevant keywords were changed to reflect this).
  – Synchronized blocks may be nested within atomic blocks. (In N3718, atomic transactions could be nested within relaxed transactions, but relaxed transactions could not be nested within atomic transactions.)
  – We decided to expand the set of functions in the standard library designated transaction-safe, and the set of exceptions that can cancel an atomic block. (This document does not fully reflect the intended change; instead, it includes relevant comments at appropriate places.)

Static Checking and removal of Safe-by-Default

This has already been approved by EWG in Rapperswil.

Safe-by-default (SBD) is the design in N3999 where the implementation is allowed to generate two versions of functions for cases where they apply and is necessary: one a transaction-safe and another that is transaction-unsafe. The implementation is allowed to choose at link time and possibly discard the unused one depending on the facility supported.

Up to now, we have exposed this consequence through several meetings, and while we have heard some saying they dislike it (while others may even say they are OK with it), none have said it is over my dead body. The EWG review in Rapperswil is the first. We have been in effect waiting to hear this, and did not want to ignore this.

Those members who objected also offered a solution which would resolve their objection. In fact, this was an earlier design before SBD and is implemented in GCC 4.7 based on N3725: Original Draft Specification of Transactional Language Constructs for C++.

They feel the SBD solution would be non-portable depending on the quality of linker (say on older VMS platforms), or whether full program analysis would be enabled, which might even depend on what optimization was turned on. This breaks the spirit of a Standard which is about portability. However, an implementation is always allowed to do more and enable optionally SBD.

This design change was approved and is reflected in the wording in N4179.

Specifically, we will require explicit annotation for non-template inline function, or
an inline function if declared without a body if it's used before the definition (likewise for templates), or"plain" extern functions. But for all other cases (specifically the templates that fall outside the above list) do not need to be annotated.

`[[optimized_for_synchronized]]` for synchronized Statement

This still requires EWG approval but is reflected in N4179.

With Safe-by-default generating two versions, there was no need for this attribute. With static-checking back in the design, we require this attribute to bring back speculation in synchronized blocks.

As with much of our design, there is implementation experience as this is an attribute which has been implemented in the Intel STM compiler but is named `transaction_callable`.

There are some library functions that could not be made transaction-safe. Examples are assert, fprintf, perror, and abort. Consider this example from memcache:

```
store_item(){ // in thread.c
  ...
  synchronized {
    ret = do_store_item(...)
  }
  ...
}
do_store_item() { // in memcached.c
  ...
  if (...)
  else if (...)
  else if (...)
    if (...)
    else if (...)
    else
      ...
     if (settings.verbose > 1) //A
       fprintf(stderr, "CAS: failure: expected %llu, got %llu\n", (unsigned
long long) ITEM_get_cas(old_it), (unsigned long long)ITEM_get_cas(it));
  ...
}
```

All the conditionals are there to show that even if the `verbose` test in Line A passes (when `verbose` is 2 or higher), it is still a very rare case that fprintf will run. However rare, `do_store_item()` cannot be made transaction-safe because it contains an unsafe code path with `fprintf`. This means that when it is called from a synchronized block from within `store_item()`, (because `do_store_item()` is not transaction-safe) it must serialize before the call to `do_store_item()`, since there is only an uninstrumented version of the function.

This is unsatisfactory for most of the normal paths of this function. So a new optimization function attribute is needed: `[[optimized_for_synchronized]]`. This indicates to the compiler that (a) `do_store_item()` might be called from a synchronized block in another compilation unit, and (b) the programmer thinks that the compiler should generate an instrumented version of `do_store_item()` even though it has unsafe code, because doing so would incur serialization only for those control flows in which the `fprintf()` is reached.

In effect, a function annotated with `[[optimized_for_synchronized]]` may have irrevocable operations and legacy function calls inside its lexical and dynamic scope. Such a function may call other similarly annotated functions within its lexical and dynamic scopes, and may contain synchronized or atomic blocks. Further, a function that is annotated as such may contain indirect function calls and virtual function calls even if it is unknown at compile-time whether the target of a function pointer is also annotated as such.

A further example demonstrates its use:

```
// translation unit 1
[[optimize_for_synchronized]] int f(int);

void g(int x) {
  synchronized {
    ret = f(x*x);
  }
}

// translation unit 2
extern int verbose;

[[optimize_for_synchronized]] int f(int x)
{
  if (x >= 0)
    return x;
  if (verbose > 1)
    std::cerr << "failure: negative x" << std::endl;
  return -1;
}
```

If the attribute were not present for `f`, which is not declared `transaction_safe`, a program might have to drop out of speculative execution in `g`'s synchronized block every time when calling `f`, although that is only actually required for displaying the error message in the rare verbose error case.

Virtual function with transaction_safe noinherit

This still requires EWG approval but is reflected in N4179.

During LWG review in Rapperswil, it was discovered that we need to consider the case for destructors for exception types (i.e., classes that derive from `std::exception`). When an exception is thrown from within an atomic block, its constructor and destructor are called within a transaction. So, by our current rules, they must be transaction-safe. However, the destructor calls the destructor of `std::exception` (because it derives from `std::exception`), so, again by our current rules, `std::exception`'s destructor must be transaction-safe, which in turn forces all classes that derive from std::exception to have transaction-safe destructors. This viral effect is unacceptable and undesirable for legacy code who might not know or care about transactional memory.

There is a similar legacy problem for virtual functions of well-established library classes in general: We cannot declare such a function to be transaction-safe, even if it is transaction-safe, because existing derived classes might not be. Thus, we cannot call such virtual functions from atomic blocks (in the current rules).

Another similar case is for function pointer parameters to functions, such as the compare argument to qsort: We cannot call qsort within an atomic block unless it is declared to be transaction-safe, in which case, because qsort may call through its compare argument, we must also declare compare to be transaction-safe. But legacy calls to qsort may have passed transaction-unsafe compare functions.

Of these three cases, we must address the first two cases, i.e. virtual destructors and virtual functions in general. For the other case (function pointer), SG5 is considering if we should use the same solution (though with a possibly more suitable name) or leave them until usage experience requires a solution. This remains open for debate.

Initial proposals to address the first case were:

(a) make a special case for destructors,
(b) make a special case for all virtual methods of std::exception, or
(c) allow transaction-safe virtual methods to be overridden by transaction-unsafe virtual methods.

Another possibility that we didn't discuss is to make an even narrower exception, just for the destructor of std::exception.

None of these proposals address the function pointer case, since we hadn't raised it yet at the time.

SG5 agreed that each of these was unpalatable, in part because it felt rather ad hoc, or, for proposal (c), that it meant that we had to forego static checking for all virtual methods, even ones without legacy issues.

We propose to adapt a variant of (c):

(d) introduce a new kind of transaction-safety declaration for virtual methods that allows them to be overridden with transaction-unsafe virtual methods. This is a new keyword called `transaction_safe noinherit`. The precise naming is still open for debate. Even within SG5, we have considered and rejected names such as

- `transaction_safe explicit`
- `transaction_safe unchecked`
- `transaction_safe maybe`

Note that all of these proposals allow transaction-unsafe functions to be called within a transaction, resulting in a run-time error. We think this is unavoidable. The proposals differ in when they allow this to happen.

There are some within SG5 who feel we should treat function pointers in the same way for the qsort case. However, there is an escape path for the qsort case. You can have two overloads of qsort: One declared transaction-safe and taking a function pointer to a transaction-safe compare function. The other not declared transaction-safe and taking a (plain) function pointer for compare.

This is actually possible because a "pointer to transaction-safe function" is a different type than "pointer to function". These two qsort overloads being totally different functions makes it possible to declare one as transaction-safe and the other as not. (Something that is currently not possible with functions that do not otherwise differ in their signature.)

We can extend proposal (d) straightforwardly by allowing function pointers to be declared `transaction safe noinherit` (possibly with a new name that better fits the context). A function pointer so declared could be assigned a transaction-unsafe function, and it could be called through within a transaction (resulting in a run-time error if it points to a transaction-unsafe function). There are others who worried that extending unchecked transaction-safety to function pointers was going too far (due to the connotation that comes with `noinherit` rather than any objection with the concept), that there would be possible pushback from the committee, and that we are conflating different issues.
(1) the problem that arises because the destructor of a derived class calls the destructor of its base class,
(2) checking that a function definition is transaction-safe, and
(3) the inheritance of transaction-safety for virtual methods (i.e., requiring that an overriding virtual method of derived class be transaction-safe if the corresponding virtual method in the base class is).

The fundamental issue that we agree we must resolve is due to issue (1) and (3) together. That is, we

can address it by addressing either (1) or (3). Proposal (a) above addresses (1); the others address issue (3).

Michael Scott suggested that we consider having two separate mechanisms (he suggested this as a conceptual exercise, not necessarily that we should actually separate them): asserting that a particular function being defined is transaction-safe (resulting in a compile-time error if it is not), and asserting that derived classes must not override a virtual method with a transaction-unsafe function. That is, the two mechanisms would check issues (2) and (3) respectively. Applied to a function pointer, the second mechanism would statically forbid the assignment of a transaction-unsafe function. Separating these mechanisms makes clear that there are two variants to proposal (d) when extended to function pointers, one that checks that the initial value of the pointer (if any) is a transaction-safe function, and the other that doesn't. The same issue actually applies to virtual methods too, because they may be pure. However, pure virtual functions may or may not have definitions.

Ignoring the naming issue (although it does affect how we think about the mechanism), we need to decide whether we prefer to have a more restrictive solution or a broader one. The restrictive solution has several advantages:

- it is easier to provide a more general solution later, if needed, than to restrict an overly broad one.
- we can get feedback about what kind of a more general solution, if any, is actually needed, and use this to inform our design.
- a more restrictive solution might be easier to implement.

However, an overly restrictive solution may cause programmers to avoid using atomic blocks. Also, if we adopt a more restrictive solution, we should consider whether it is compatible with a future broader solution. In particular, if we make a special case for destructors, then that mechanism might not be a special case of proposal (d), and it may introduce unnecessary complications. (unnecessary because if we adopt proposal (d), we don't need a different special case for destructors.) But if we choose a restrictive solution that is simply the broader solution but applied only to some cases, programmers might find the restrictions arbitrary and counterintuitive.

As a possible extension, proposal (d) extended to function pointers is intended to be a single mechanism that happens to address the destructor problem, but it does so as part of a broad straightforward change, rather than one tailored for that problem. It simply says that we can declare a function pointer that can be called through within a transaction, but isn't actually checked for transaction-safety. This declaration is a gesture we force the programmer to make to take responsibility for the possible run-time error. In that proposal, we also require that the initial value of the function pointer is transaction-safe: if that value is a transaction-unsafe function, the program is rejected by the compiler. However, later assignments to such function pointer are not checked. Some disagree with providing this facility and prefer the same checking for the initial value and for later assignments, to avoid ugly workarounds of this rule. Applied to virtual methods, this means that if the class declares a virtual method to be unchecked transaction-safe, and also defines it (i.e., it is not a pure virtual method), then the definition

is checked and rejected at compile time if it is not transaction-safe. However, overriding functions of unchecked transaction-safe virtual methods in derived classes are not checked (unless the derived class explicitly declares them transaction-safe or unchecked transaction-safe).

For this paper, SG5 agreed for now is to adopt a restricted version of proposal (d), or actually the original form of proposal (d), not extended to function pointers.

So a function that overrides a function declared `transaction_safe`, but not `transaction_safe` **`noinherit`**, is implicitly considered to be declared `transaction_safe`. Its definition is ill-formed unless it actually has a transaction-safe definition. A function declared `transaction_safe noinherit` that overrides a function declared `transaction_safe` (but not `transaction_safe noinherit`) is ill-formed.

Here is an example demonstrating `transaction_safe noinherit`:

```
struct B {
  virtual void f() transaction_safe;
  virtual ~B() transaction_safe noinherit;
};

// pre-existing code
struct D1 : B
{
  void f() override { }   // ok
  ~D1() override { }   // ok
};

struct D2 : B
{
  void f() override { std::cout << "D2::f" << std::endl; }
       // error: transaction-safe f has transaction-unsafe definition
  ~D2() override { std::cout << "~D2" << std::endl; }     // ok
};

struct D3 : B
{
  void f() transaction_safe noinherit override;
         // error: B::f() is transaction_safe
};

int main()
{
  D2 * d2 = new D2;
  B * b2 = d2;
  atomic_commit {
    B b;        // ok
    D1 d1;      // ok
    B& b1 = d1;
    D2 x;       // error: destructor of D2 is not transaction-safe
    b1.f();     // ok, calls D1::f()
    delete b2;  // undefined behavior: calls unsafe destructor of D2
  }
```

```
}
```

Add Transaction-safety for all containers and iterator-related functions

This still requires LEWG approval but is reflected in N4179.

This is applied in N4179 to unique_ptr, strings, array, deque, forwardlist, list, vector, vector<bool>, map, set, multiset, unordered_map, unordered_multimap, unordered_set, unordered_multiset as well as iterators.

Add tx_exception

This still requires LEWG approval but is reflected in N4179.

We intend to introduce a

```
template<class T>
class tx_exception : exception { ... };
```
with a transaction-safe "what()" function and where "T" can be memcpy'd.

This means that a specialization of `tx_exception` supports transaction cancellation only if `T` is a trivially-copyable type.